ADA* EVALUATION PROJECT

THE DEVELOPMENT OF ADA* SOFTWARE FOR SECURE ENVIRONMENTS

Prepared for

HEADQUARTERS UNITED STATES AIR FORCE
Assistant Chief of Staff of Systems for Command, Control,
Communications, and Computers
Technology & Security Division

Prepared by
Standard Automated Remote to AUTODIN Host (SARAH) Branch
COMMAND AND CONTROL SYSTEMS OFFICE (CCSO)
Tinker Air Force Base
Oklahoma City, OK 73145
AUTOVON 884-2457/5152

* Ada is a registered trademark of the U.S. Government
(Ada Joint Program Office)

23 May 1986

THIS REPORT IS THE THIRD OF A SERIES WHICH DOCUMENT THE LESSONS LEARNED IN THE USE OF ADA IN A COMMUNICATIONS ENVIRONMENT.

ABSTRACT

This paper discusses software security and seeks to demonstrate how the Ada programming language can be utilized as a tool to implement software design methodologies which support software security.

The major security risk in the military-telecommunications environment is the compromise of secure or sensitive information and/or not delivering a message or part of a message. Software security issues intended to eliminate these and other security risks are numerous. This paper addresses a limited number of issues to illustrate how Ada is being used to accomplish a more secure software product. Security issues related to interlacing message data, prevention of lost data, message and command validation, message distribution integrity, and information protection are addressed.

The paper goes into a description of how the SARAH designers are approaching the problem of designing for a secure environment. Good software engineering practices need to be applied if effective software is to be developed for secure systems. Some of the areas which need to be addressed during the analysis/design phase include localization, understandability, reliability, abstraction, confirmability, and modularity.

Ada provides a rich set of language features which can be used to develop reliable, survivable and secure software systems. The Ada language was designed to support the development of large systems which would be developed using modern software engineering principles. The features that directly support the development of software for secure systems are strong data typing, packaging, generics, and exception handling.

Other aspects of the Ada environment that must be considered in the security of the final product are the newness of the technology, differences in implementation techniques by the variety of vendors in the market, and new programming support tools that have little or no track record to verify the stability of the product.

Ada Evaluation Report Series by CCSO

Ada Training	March 13, 1986
Design Issues	May 21, 1986
Security	May 23, 1986
Module Reuse	Summer 86
Micro Compilers	Fall 86
Ada Environments	Fall 86
Transportability	Winter 86-87
Modifiability	Winter 86-87
Runtime Execution	Winter 86-87
Testing	Spring 87
Project Management	Spring 87
Summary	Spring 87

TABLE OF CONTENTS

Appendices

LIST OF FIGURES

1. INTRODUCTION

1.1. BACKGROUND

Today's world political and military situation is such that compromise or interception of classified or sensitive military information could have grave consequences for our national security. Stringent methods of data protection are required. There are a number of ways of protecting classified and sensitive information in the Automatic Data Processing (ADP)/Telecommunications environment[1]:

- Physical Security: The prevention of uncleared personnel from gaining physical access to the controlled space around ADP/Telecommunications equipment which process classified data.

- Hardware Security: Usually involves the use of TEMPEST approved equipment to prevent the occurrence of compromising emanations. In addition to emanations, certain hardware features may work in conjunction with the system software to ensure process integrity.

- Cryptographic Security: The process of encrypting text before transmission on uncleared circuits to render the information unintelligible to potential interceptors.

- Software Security: The use of the system software to implement and enforce an array of security measures used to prevent the compromise of classified or sensitive data from ADP/Telecommunications environment.

This paper discusses software security and seeks to demonstrate how the Ada programming language can be utilized as a tool to implement software design methodologies which support software security methods.

This paper is one in a series which seeks to help potential Ada developers gain practical insight into what is required to successfully develop Ada software. With this goal in mind, Air Staff tasked the Command and Control Systems Office (CCSO) with evaluating the Ada language while developing real-time digital communications software. CCSO chose the Standard Automated Remote to AUTODIN (Automatic Digital Network) Host (SARAH)[2] project as a basis for this evaluation. SARAH is a small to medium size project (approx. 40,000 lines of source code) which will function as a standard-intelligent terminal for AUTODIN users and will be used to help eliminate punched cards as a transmit/receive medium. The development environment for SARAH consists of the SofTech Ada Language System (ALS) hosted on a Digital Equipment Corporation VAX 11/780, a Burroughs XE550 Megaframe and several IBM compatible PC-XT and PC-AT microcomputers. The ALS is the focal point of this integrated development environment. The source code developed on the XE550 and microcomputer workstations is maintained by the ALS configuration control system. In addition, reusable modules are baselined by the ALS and maintained on the VAX in a software repository.

1.2. PURPOSE

The purpose of this paper is to:

- o Outline Automatic Data Processing (ADP) security risks which may be managed through the software implementations.
- o Identify software application and design methods which can be utilized to support software oriented risk management.
- o Identify Ada language features and characteristics which support these design methodologies.
- o Caution the implementor about unknowns of the language and other considerations as they pertain to security.

1.3. SCOPE AND CONSTRAINTS

The scope of this paper is limited to those security requirements applicable to the SARAH system. SARAH, as a General Service (GENSER) system, is not subject to as many security requirements as the Defense Special Security Communications System (DSSCS)[3]. Since SARAH will be a telecommunications system, security risks discussed are those of concern to developers and implementors of telecommunications systems and may not be applicable to ADP systems. The security risks discussed are not intended to be comprehensive, even for a telecommunications system, but were selected as issues of high concern to telecommunications systems and as good examples for demonstrating the potential of the Ada language to support software security design methods.

Since the authors of this paper are currently involved in the design phase of the SARAH Ada development project, they are not in a position to advise about Ada security applications from personal experience. They have followed the development of the language, have studied the language extensively, and have a good deal of software development experience with military-telecommunications systems using a variety of languages and development methodologies.

2. SECURITY ISSUES

In the military telecommunications environment, the overlying security risk is the compromise of sensitive or classified information and/or not delivering a message or part of a message. With this in mind, we will discuss the following security issues: prevention of interlacing of message data, prevention of lost data, message and command validation, message distribution integrity, and information protection. The SARAH workstation will provide a secure communications environment for the creation, transmission, reception, and delivery of messages within the Defense Communications Agency (DCA) AUTODIN system. Issues addressed are in the context of the application for the SARAH workstation.

In this section we will look at some general security issues. In later sections we will look at both design issues and the Ada language to see their relation to security.

2.1. PREVENTION OF INTERLACING MESSAGE DATA

Message interlacing occurs when part or all of one message becomes mixed with another message. When this occurs, the interlaced message content may be of greater classification than the classification specified in the message header. This message may or may not also be delivered normally, depending on what caused it to become interlaced with the other message. Messages are split into blocks at various points during transmission to facilitate efficient handling. The system handles any number of messages at one time and must maintain each as an entity. Each message entering the network contains a message header which consists of a number of information fields. The classification field is used to prevent a security compromise. Other fields within the header and trailer are used to check for proper length and ending block information.

2.2. PREVENTION OF LOST DATA

Lost data results in message data not being delivered as intended. To prevent this, the SARAH software must maintain system integrity (integrity, as used in this paper, means that independent components or entities remain independent and complete) and data flow integrity at all times.

One way to prevent lost data is to ensure all information entered into the SARAH terminal, both control information and message information must be validated, usable, and appropriately handled. Any introduction of information that is subsequently never used or inaccessible is unwarranted and should be prevented.

Another thing that must be done is to ensure the message traffic is properly queued. Improper queuing of messages for delivery through the system can result in either lost messages (undelivered) or misrouted (delivery to the wrong customer). Either case must be prevented.

To prevent data from being lost, the distribution process must cover all contingencies. Initialization of the output routing matrix must result in specific instructions for handling a message whose address does not match any of the entries in the matrix. Since any work station can serve as a backup for another while receiving its own traffic load, this default queue logic must be available.

The final ingredient in preventing lost data is to maintain appropriate and sufficient information about messages being processed. This will allow recovery of all undelivered traffic after a failure of the system. The SARAH system will satisfy this requirement with a printed journal and various techniques for recovery from floppy disks and the host computer system. These steps will ensure no messages are lost during the recovery process.

2.3. VALIDATION

2.3.1. COMMAND VALIDATION

The system menu function takes commands from the keyboard and directs the command selections to the other system functional units (or modules) of the system. Each incoming command is validated by the menu function to ensure the command is a member of a limited set of commands allowed for each of the other system functional units.

2.3.2. MESSAGE VALIDATION

Before entry into the AUTODIN network, the message will be validated by the message preparation module of the SARAH system. This validation will check the contents of the message, keying on the message header, to ensure that it meets all the criteria for proper routing in the AUTODIN network. Checking is done on precedence classification, content indicator code, date-time-group, originating station routing indicator, station serial number, and routing indicator codes. If an error is detected, the message will not be allowed to be stored on the message transmission disk. The operator will be notified so the message can be changed to the proper format for transmission.

2.4. DISTRIBUTION INTEGRITY

SARAH workstations will ensure that messages are delivered to the proper addressee by validating message header information against an output routing matrix. The SARAH system will provide a facility to create the customer output routing matrix. The matrix will be created on site at installation time and may be moditied as required. It will provide all necessary information to validate each message before delivery to the customer. Validation criteria will include classification identification, routing indicator code, distribution media (printed and/or floppy disk), etc.

2.5. INFORMATION PROTECTION

Information protection, in this context, is the isolation of system data bases (both the information in and the processing logic of the data bases) from unexpected effects of messages being processed or other dynamic data. One area of concern is the message queuing systems which control the order of delivery of messages being transmitted, received, or being prepared for distribution. These queues may have to interact with a system data base (i.e. customer distribution matrix) but not be allowed to affect the data base. The system must be equipped with appropriate safeguards to ensure information protection. As we will see later, the Ada language has some features that will help us.

3. DESIGN FOR SECURITY

As the size and complexity of software systems increases, so does the probability of error. For secure systems, errors in coding or design could have disastrous effects on national security. Good software engineering practices need to be applied if effective software is to be developed for secure systems. Some of the areas which need to be addressed during the analysis/design phase include abstraction, modularity, localization, understandability, confirmability, and reliability.

We will now take a look at several design issues that can impact security. We will not specifically look at the Ada language until the next section of the report.

3.1. ABSTRACTION

Abstraction should be used when designing secure systems so that security issues are not clouded by less relevant details. Abstraction is one of the fundamental tools for controlling complexity[4]. We use abstraction in our lives every day to control complexity; the principles of abstraction for software engineering are no different. For example, when we look at a computer system, we see the visual display, keyboard, processor and storage devices. In effect, we have abstacted our view of the computer. We know that the processor is made up of integrated circuits, decoupling capacitors etc.; however, those aspects are not relevant to our top level view and if introduced would create an unnecessary amount of complexity. Perhaps so much so that we would forget about the functionality of the major components.

The principle of abstraction is being extensively used for the SARAH software system. Figure 3.1 shows the top level abstraction view of the SARAH software. At this level, the SARAH designers knew that several requirements needed to be incorporated into the system. For example, connection to the Defense Communications System was a requirement, and so a subsystem that handled communications (Comm) was needed. At this level of abstraction, the designers knew that other elements of the system were also needed such as a message edit/preparation facility and an input/output (I/O) system.

At this level of abstraction, high level system interfaces and overall system functionality can be defined. Working at such a high level of abstraction, the designers have been able to address high level security issues. For example, the Comm system will operate independently of the Edit system and not rely on other systems to maintain queues or manage the incoming or outgoing messages.

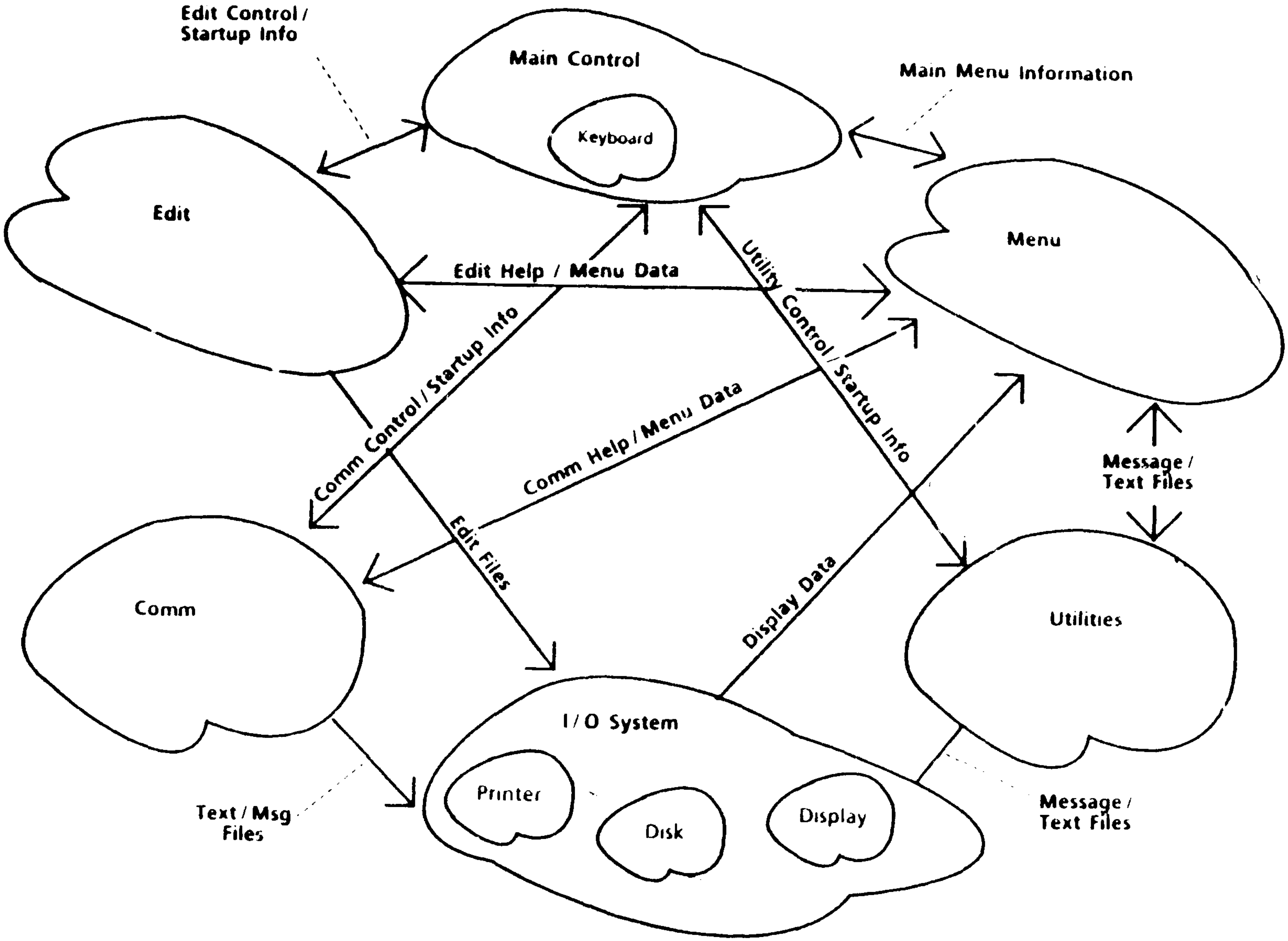

Figure 3-1. SARAH Top Level Topographical Chart

One of the major security requirements for the SARAH system is that the transmit and receive data must not be interlaced. As such, when the next level of abstraction was defined for the Comm system, two separate and independent modules were identified. Using successive levels of abstraction, the SARAH designers have been able to control the complexity of the system and focus on the security aspects relative to the level being defined.

To consider all aspects of security at one level can create a situation where some aspects of security may be overlooked. The control of complexity is extremely important for the development of software for secure systems. As shown, the application of abstraction to software design can help control this complexity and help incorporate security requirements into the resulting software.

Another software engineering principle that goes "hand-in-hand" with abstraction is information hiding. Whereas abstraction is used to extract the major information at a particular level in a design, information hiding makes certain functions and data inaccessible to elements of the design which do not need them.

The Hierarchical view of SARAH's top abstraction level (Figure 3.2) provides a good example of this software engineering feature. For example, the start up module of the Comm System is visible to the Main controller; however, the Transmit and Receive tasks are not. As such, the Main controller cannot use any of the functions or data within the transmit or receive tasks. Similarly, although Print Mgr is visible to Comm, the converse is not true and so Print Mgr cannot use or manipulate any of the Comm data or functions. Information hiding is therefore a very powerful mechanism for protecting data within secure systems.

3.2. MODULARITY

A major requirement for software system design is modularity. Modularity deals with the decomposition of a system into a number of constituent parts which are interfaced together. Modularity is a fundamental tool for helping us manage the complexity of software systems. A good modular design will purposely reflect the real world problem space. The modules that come out of this type of design are usually writ en separately and later joined to form the program.

Ideally, the modules should exhibit loose coupling. This can be achieved by keeping the module interfaces to a minimum and collecting or localizing like data and functions. Software developed for secure systems should be modular and the modules should have loose coupling so programmer induced errors can be localized, and complexity can be controlled.

A software system that has well defined interfaces between modules will reduce the possibility of programmer induced errors during maintenance. If couplin, is high, a change to one module could inadvertently change the function of another module. Even worse, the system data may be unknowingly modified and so cause a security compromise. The Ada language provides features which aid in making software transportable. As such, software systems developed with Ada could be in service for quite a long period of time (perhaps in excess of 20 years). Having modules that are long lived means that the modules are well tried and proven. Well defined modules which exhibit low coupling can effectively reduce maintenance problems and so help maintain system integrity throughout the life of the software.

3.3. LOCALIZATION

Localization deals with the proximity of like entities. A module that has strong cohesion will be very localized because it will encapsulate like functions and data. For example, the Display Tools Package in Figure 3.2 exhibits a high degree of localization. The package contains a number of tools for outputting Display Data to the screen. The only operations that can be performed on display data are those provided by the package.

For a secure software system, localization is extremely important. In the SARAH system, distribution integrity and message interlacing are major security concerns. Localization can provide the basis for eliminating these security risks. By localizing data and functions to specific modules or packages (we'll see how Ada can help us do this, later), we can prevent inadvertent data modification or manipulation. For example, Received Message Data and Transmit Message Data cannot be intermixed because they are of different data types encapsulated in separate modules which have their own controls for data manipulation.

3.4. UNDERSTANDABILITY AND CONFIRMABILITY

If a software system is not understandable, the possibility of a security compromise could be high. Understandability relates not only to the resulting code, but also to the overall design. For example, if the software solution does not reflect the real world problem, the design could appear excessively complex to maintenance personnel. The maintainer may therefore make a change to the system without fully knowing the overall effects of the change.

At the code level, a defined coding style should be used so that the software is readable and therefore understandable. The low level design/implementation standard should be well defined prior to development. For the SARAH project, this standard is based on a style guide published by Intellimac Inc.[5] and is included in the SARAH Software Standards and Procedures Manual (SSPM) as required by the DOD-STD-2167, the documentation standard for the project.

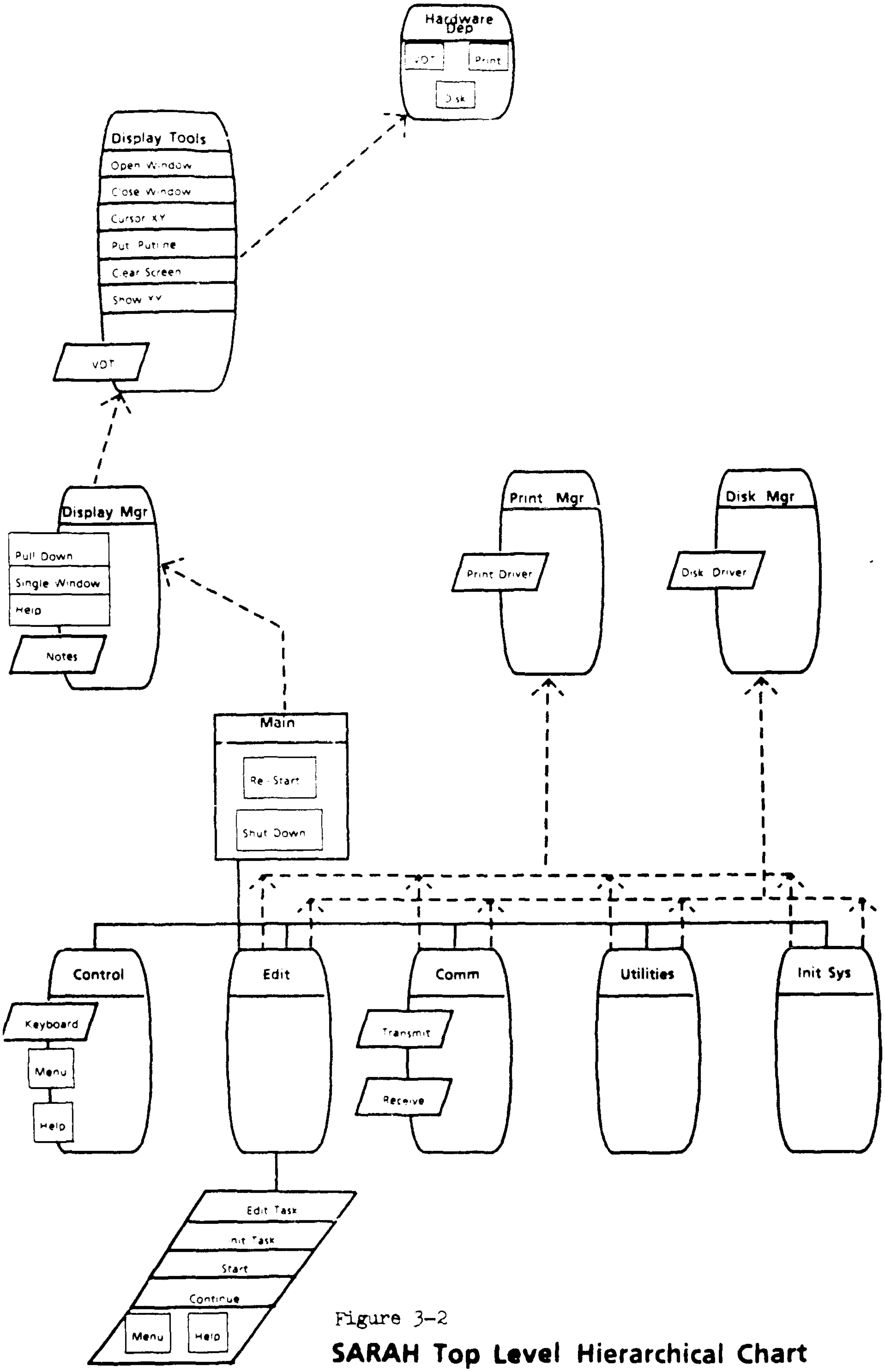

Figure 3-2

SARAH Top Level Hierarchical Chart

A defined style has a number of positive effects on the ability of software to reliably process sensitive information. First, if the software is understandable, then functionality is more easily validated and confirmed during walkthroughs and reviews. Second, a defined structure and style promotes completeness. For example, if the SSPM requires that all types, exceptions, and objects are to be commented, the programmer is forced to consider the declarations more carefully. In addition, the maintenance programmer will have the benefit of these comments when the software is in its maintenance phase.

3.5. RELIABILITY

Reliability of software for secure systems is of utmost importance. An unreliable system which processes secure information could have drastic effects on national security. For example, if under certain conditions, a message system allowed classified messages to go to an unclassified local delivery point, a serious compromise would occur. One of the major design goals of the SARAH development has been reliability.

Reliability cannot be added on after the system has been developed; rather, it must be designed into the system. Risk analysis should be completed at each design abstraction so that possible problems are not overlooked. Unfortunately, the most popular software lifecycle models (e.g. Waterfall Model) are not risk driven and so do not provide a formalized basis for risk analysis. To overcome this deficiency, the SARAH designers decided to borrow heavily from the Spiral Model[6] and introduce risk analysis at the completion of each design cycle.

4. USE OF ADA LANGUAGE FEATURES

Ada provides a rich set of language features which can be used to develop reliable, survivable and secure software systems. These features directly support the modern software engineering principles discussed in the previous section. This is not surprising since Ada was designed to support the development of large systems which would be developed using these engineering principles. Some of the Ada features that directly support the development of software for secure systems are strong data typing, packaging, generics, and exception handling.

4.1. STRONG DATA TYPING

Ada is a very strongly typed language. Strong data typing provides the basis for protecting data objects within the system. Some of the types that can be used to produce effective secure software include private types, enumeration types, and access types. In addition, Ada provides the programmers with the ability to define their own types. These user defined types can be extremely effective for the development of secure systems.

One of the security issues being addressed in SARAH is message validation. Ada provides powerful typing features which help accomplish message validation. For example, by using user defined types together with enumeration types, the programmer can strictly define the scope of a particular object. When the message header is validated, only certain security classifications will be allowed. In the header, only the first letter of the classification is provided. For example, Top Secret is identified as "T". A typical Ada type declaration for security classifications is:

```
type CLASSIFICATIONS is (T, S, C, R, E, U);
```

Suppose we want to validate the message headers for correct security, then we need to create an object of this type to which we can then make assignments. This can be done as follows:

```
Header_Security : CLASSIFICATIONS;
```

If, during execution, a character is assigned to Header_Security from the incoming message, and it is not in the range specified by the user defined type (CLASSIFICATIONS), then Ada will flag this as an exception condition and the software can take the appropriate measures to handle this problem.

The use of enumeration and user defined types are also very effective for validating user input commands. A type can be defined so that only a limited range of input will be accepted. Security is improved because the resulting software is more understandable and reliable.

4.1.1. Private Types

Another Ada typing feature which can help in producing secure software systems is private types. Private types are very powerful for protecting data. If a private type is declared in an Ada package, then the only operations that can be performed on objects of that type outside its package are equality/inequality tests, assignment, and any operations declared in the package specification. If even more protection is required, a limited private type can be used which restricts the package user to only those operations shown in the package specification. For limited private, even tests for equality/inequality and assignment are not permitted.

In the SARAH system, Transmit_Data and Received_Data are limited private types encapsulated in separate packages. If another package needs to access this data, for example a validation package, then the only operations that can be performed are those listed in the package specifications. The validation package can look at certain aspects of the incoming message; however, it cannot manipulate or make a copy of the data. Visibility between packages is controlled and this further reduces the possibility of unlawful access of secure data. For example, the receive and transmit packages are completely independent and are not visible to one another. As such, there is no possibility of mixing (or interlacing) Received_Data with Transmit_Data. As shown, private types are a very powerful mechanism for protecting and restricting access to secure data within a software system.

4.1.2. Access Types

Access types provide a powerful mechanism for creating secure dynamic lists and queues. Queuing in communications systems is an important function. Queue implementation using older languages and assembly languages is a complex problem, particularly when there are a large number of transmit and receive lines. As such the possibility of misrouting information is increased. Access types provide a well defined mechanism for implementing queues, and so the resulting implementation is more readable and understandable. The software can therefore be more easily verified, and so reduce the risk of a possible compromise situation.

Since access types can be made private or limited private, the benefits of data protection are also available to the designer. A standard queue type can be defined and several independent objects of this type can be created. For example, in a Receive package, a queue may be required for each local delivery point. Each of these delivery points could be objects of the queue type. If the queue type is made private, the actual data structure is hidden from the user. In addition, if limited private is used, the only operations that can be performed on the queue objects are those specified in the visible part of the encapsulating package. In summary, access types provide a mechanism for easily defining queues and lists.

4.2. PACKAGING

One of the unique features of Ada is the package. An Ada package is made up of a visible part and a body. The visible part contains the package specifications and shows which entities can be exported. For example, a package specification for a package of display tools may tell the user that the operations which can be performed on Display_Data are Open_Window, Close_Window etc.; but the details of how this is done is hidden in the package body. As such, packages provide a powerful method of implementing abstraction and information hiding. As discussed earlier, these are extremely important concepts for the design of secure software.

Packaging concepts are being used extensively in the SARAH project for controlling complexity, protecting data, promoting understandability and enhancing reliability. Figure 3.2 shows how the SARAH software has been packaged. Each of the major functional modules are implemented as packages. These packages have defined interfaces to other packages. In addition, the package dependencies are clearly shown. By packaging for low coupling and high module cohesion, package dependencies are minor. For a secure system, this is important because if a problem arises in one module, the effect on other modules will be negligible.

4.3. GENERICS

Generics provide a number of desirable benefits for secure software implementations. Generics are templates of Ada program elements. For example, several instances of a generic package can be made by simply instantiating (or filling-in) with different data types. In SARAH, a Variable List package is being created to provide a mechanism for queuing and list processing. The package will be written just one time and then be instantiated for the various other data types. For example, a queue will be created for Received_Headers and Transmit_Headers. In addition, the same package will provide variable list facilities for the text processing section of the system. The productivity benefits of generics are substantial; however, generics also have a positive effect on system reliability, modularity, and understandability.

As mentioned earlier, queuing software for older communications software is complex, and not very understandable to the inexperienced. By using generics, we will reduce the complexity considerably because there will be only one major functional module and this one piece of code will be instantiated to provide functionality for several independent objects. In essence, a package will be created for each data object. The resulting software will theretore be more modular to help enhance understandability. The verification process for SARAH will be considerably easier and more complete because of generics.

Software reuse has a direct effect on reliability and can aid in reducing the risk of compromise in secure systems. Generics provide an excellent method for software reuse. Reliability is enhanced through the use of generics because a previously verified module is less likely to contain programming errors. Indeed, even with older languages such as FORTRAN, where mathematics libraries were extensively reused, few errors resulted from the reused code.

4.4. EXCEPTION HANDLING

Ada provides a feature called exception handling to catch errors during program execution. This feature is easy to use and makes the resulting software very readable, reliable, and survivable. As such, exception handling is an important concept for the design and implementation of secure software. Extensive use should be made of exception handling to reduce the possibility of a security compromise caused by unreliable software or hardware.

As previously indicated, reliability must be built into the system from the outset. When analyzing and designing each level of abstraction, the designers should identify and record exception conditions which could possibly create security problems. In particular, these conditions should be identified when risk analysis is performed following the completion of a design abstraction level. These exception conditions can then be accounted for by Ada exception handlers.

5. FURTHER CONSIDERATIONS

Ada is opening a whole realm of possibilities for improved software systems development. Caution must be exercised, however, when entering this new domain. There are negative aspects of software development using Ada that any new developer should be aware of and be prepared to address. The "catch-22" is basically the newness of the language and the proliferation of compilers and Ada accessories available on the commercial market. Unknown problems may lie within the tens of thousands of lines of code comprising the compilers from each vendor. These problems, undetected through the validation procedure, are there and will have to be addressed on an "as occurs" basis. Due to the diversity of the compiler designers in the Ada world, solutions to the same problems in language implementation take on different designs. This difference in understanding and solution processes makes for unknown numbers of future problems. Effects of this on the concepts of data typing, module isolation, limited access, etc. are risks that have to be taken and faced when and if problems do surface.

Another area of risk that should be highlighted is that of the support environment that is available with the Ada compilers being acquired today. The Ada Programming Support Environment (APSE) is a set of development tools that can assist and provide added control over the software being developed. Some vendors are marketing support environments that come from existing product lines and are proven. Tools such as Ada syntax checkers, debuggers, stub generators, code generators, etc. are available and being billed as wonderful. Care must be taken to ensure the proper function of such tools. The assumption that the resultant code is pure and secure should not be made. Vigilance is necessary in this emerging Ada world of technology and we must be the front line in preventing the occurrence of problems.

6. SUMMARY AND RECOMMENDATIONS

6.1. SUMMARY

Modern software engineering techniques and specific features of the Ada language provide very useful tools for use in developing secure ADP and telecommunications systems. The most dangerous risk to these systems is the compromise of classified or sensitive information and the loss of important information.

The SARAH workstation project, as a telecommunications terminal, must address a number of security issues; among them are the prevention of interlacing of message data, the prevention of lost message data, system command and message validation, distribution integrity, and information protection.

The major software design methodologies useful for handling these issues are abstraction, modularity, and localization. Abstraction allows the designer to concentrate his/her efforts on a particular problem. Modularity and localization allow the system functions to be logically grouped and provides for more easily controlled interfaces.

These methodologies also seek to achieve the goal of understandability, verifiability, and reliability. Reliability is greatly increased by designing a verifiable system which can be completely and easily tested. Systems must be understandable, both at the design level and at the code level. Understandability builds reliability into the system; it also helps to ensure that the system will continue to be reliable after maintenance. Maintenance programmers need an understandable system to help them locate problems, correct problems, and predict the effect of modifications to the system.

Ada provides features that support modern design methodologies and thus software security. Strong data typing and exception handling helps to ensure the system will function reliably and predictably when subjected to many different input conditions. Packaging is one of Ada's most useful features; it provides an efficient mechanism for modularization, localization, abstraction, modifiability, and understandability.

Caution should be exercised because of compiler and support environment unknowns. Ada is a large language and various commercial compilers implement the language features in many different ways. Ada has arrived with many support environment tools whose efficiency and predictability may vary.

6.2. RECOMMENDATIONS

o Become aware of modern software engineering principles and prepare to enforce their application to all aspects of software development.

- Use the principles of abstraction extensively in the area of localization and data typing to ensure a more secure and maintainable software product.

- Make extensive use of exception handling and apply it throughout the software system to ensure a stable and controlled software environment.

- Be aware of which Ada compiler you are using, and make sure you keep current as improved and validated versions become available.

A. REFERENCES

[1] Automatic Data Processing (ADP) Security Policy, Procedures, and Responsibilities, Air Force Regulation 205-16, US Air Force.

[2] "SARAH Operational Concept Document", US Air Force, 12 May, 1986.

[3] AUTODIN I System Functional Specification, Defense Communications Agency, Code 250, Washington, D.C., 1981.

[4] Booch G., Software Engineering with Ada, Menlo Park, CA:Benjamin/Cummings Publishing, 1983.

[5] M. Gardner, N. Brubaker, et. al., Ada Style Manual, Intellimac, Inc.

[6] Boehm B. W., "A Spiral Model of Software Development and Enhancement", TRW Defense Systems Group.